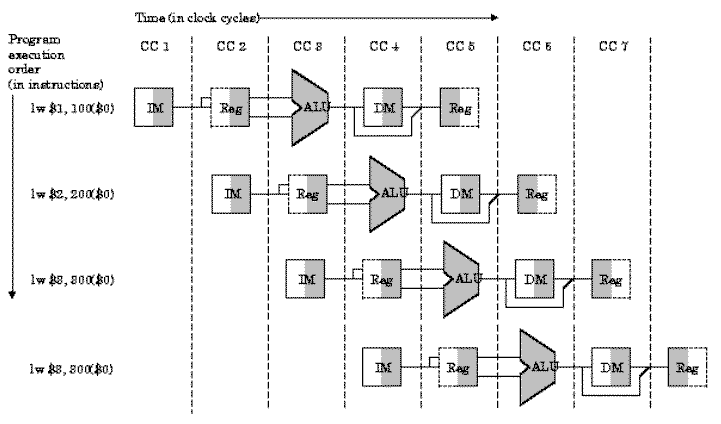
Miguel Angel Triana Camacho

Pipelined MIPS Implementation + Hazard Control

September 11th, 2024

**PIPELINED MIPS:**

The pipelined MIPS adds a complex layer to the single cycle MIPS implementation. This layer is used for improving the maximum frequencies at which the core can run. The principle of a pipeline is basically like Henry Ford did to the assembly line of automobiles. The process is split up in similar-in-length stages, this way you can increase the throughput. 

The implemented MIPS was split in 5 pipelines:

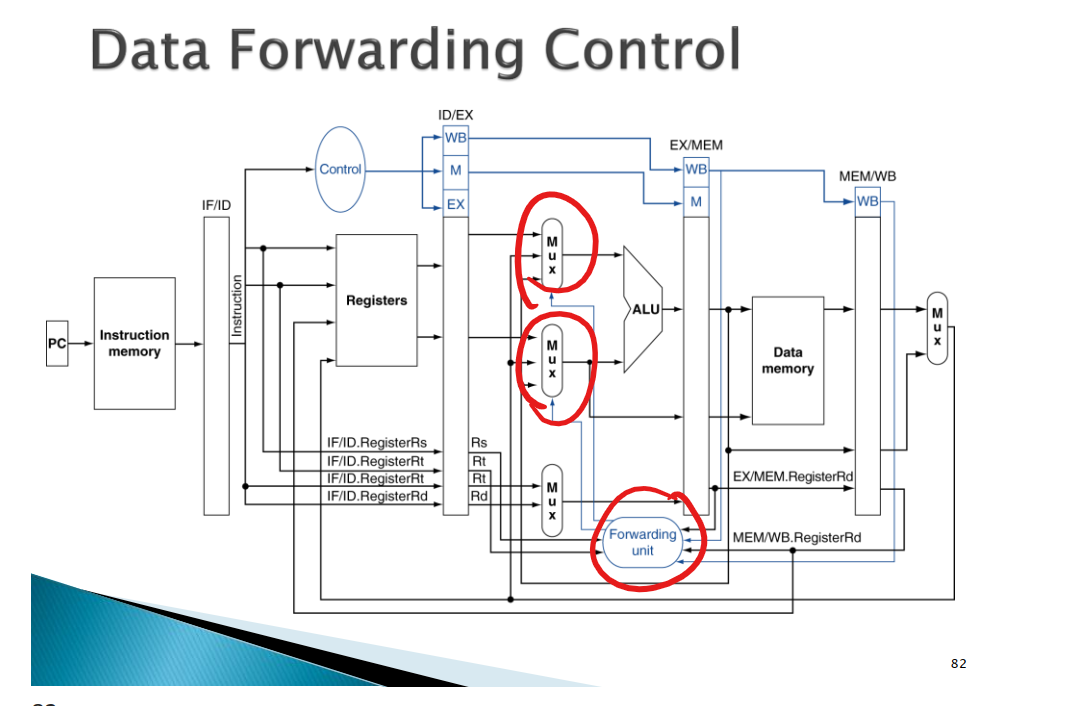
* Fetch
* Decode
* Execute
* Memory
* Writeback

In theory this would mean that the IPC would increase 5 times! But no, in reality there are several challenges to address, called hazards.

Hazards are scenarios in which the processor’s data can be corrupted due to the way the pipeline works. To avoid this, some logic needs to be implemented. Unfortunately, this logic causes the pipeline to lose some of the performance due to flushes, and stalls.

In this MIPS implementation there are several units to avoid this hazards.

The first one is the **data\_fwd,** which is used to forward data from different stages in the pipeline, to avoid waiting, or stalling, until the processor stores the results in the register bank.



Muxes are implemented to feed the ALU/Multiplier, and “bypass” the data directly from any other stage. This unit does not impact performance.

There is also a flushing unit, and a branch detection unit that together pushes branch detection one cycle earlier than execute (decode) this means that we only lose one cycle per missed branch instead of two. This unit flushes said instructions mistakenly fetched during branches (conditional or unconditional), as the processor assumes a branch **not** taken path.

There is a multiplier that takes 4 pipelined cycles to finish. This induces a lot of extra hazards that are taken care by the stalling unit. These hazards are explained below.

The biggest issue with the multiplier is that it extends the duration of the 5 stage pipeline, into a 7 stage pipeline. This brings all sorts of challenges. The first one is for dependent sources. If there are source dependent instructions in the following 3 cycles after a multiply is issued, then we need to stall to make sure that the instructions get the solved multiplication.

Second case, as the pipeline is longer, we can have write port structural issues. This can be solved by adding a second write port to the register bank, but we decided to implement a stalling solution that stalls the pipeline until the multiplication engine release the write port.

Third case is destination dependance. If there is a multiplication, and then another instruction that writes to the same destination as the multiplication, then this needs to be stalled, as without a stall it will get written before the multiplication, and in the multiplication will then overwrite the entry, breaking thus program coherency.

**Random MIPS Hazard Finder Code:**

This code tries to find issues with the MIPS pipe implementation. This is no match for our MIPS implementation.

ASM CODE:

nop

lw $0, 0($1)

lw $2, 32($1) ## Memory is byte alligned

lw $4, 36($1) ## Memory is byte alligned

add $2, $2, $4

and $3, $2, $4

lw $5, 40($1)

or $7, $5, $0

lw $6, 8($1)

or $7, $7, $6

slt $10, $0, $7

slt $11, $7, $0

addi $31, $0, -1

beq $7, $0, FAR

j LABEL1

LABEL2:

beq $0, $0, LABEL2

LABEL1:

j LABEL2

FAR:

addi $zero $zero, 1

Binary Code Assignation:

  uut.InstructionMemory.regData[0]  = 0;

  uut.InstructionMemory.regData[1]  = 32'h8c200000;

  uut.InstructionMemory.regData[2]  = 32'h8c220020;

  uut.InstructionMemory.regData[3]  = 32'h8c240024;

  uut.InstructionMemory.regData[4]  = 32'h00441020;

  uut.InstructionMemory.regData[5]  = 32'h00441824;

  uut.InstructionMemory.regData[6]  = 32'h8c250028;

  uut.InstructionMemory.regData[7]  = 32'h00a03825;

  uut.InstructionMemory.regData[8]  = 32'h8c260008;

  uut.InstructionMemory.regData[9]  = 32'h00e63825;

  uut.InstructionMemory.regData[10] = 32'h0007502a;

  uut.InstructionMemory.regData[11] = 32'h00e0582a;

  uut.InstructionMemory.regData[12] = 32'h201fffff;

  uut.InstructionMemory.regData[13] = 32'h10e00003;

  uut.InstructionMemory.regData[14] = 32'h08100010;

  uut.InstructionMemory.regData[15] = 32'h1000ffff;

  uut.InstructionMemory.regData[16] = 32'h0810000f;

  uut.InstructionMemory.regData[17] = 32'h20000001;

**Snippets:A black background with colorful lines and numbers

Description automatically generated with medium confidence**

ASM Code with the observations:

MEM is already initialized, as well as register $1 with the 0x10010000 address for pointing to memory:

  // DATA MEMORY PRELOAD

  // EDIT LINES BELOW FOR PRELOADING THE MEMORY

    uut.DataMemory.data\_mem\_ff[0]   = 10;

    uut.DataMemory.data\_mem\_ff[1]   = 9;

    uut.DataMemory.data\_mem\_ff[2]   = 8;

    uut.DataMemory.data\_mem\_ff[3]   = 7;

    uut.DataMemory.data\_mem\_ff[4]   = 6;

    uut.DataMemory.data\_mem\_ff[5]   = 5;

    uut.DataMemory.data\_mem\_ff[6]   = 4;

    uut.DataMemory.data\_mem\_ff[7]   = 3;

    uut.DataMemory.data\_mem\_ff[8]   = 2;

    uut.DataMemory.data\_mem\_ff[9]   = 1;

    uut.DataMemory.data\_mem\_ff[10]  = 5;

    uut.DataMemory.data\_mem\_ff[11]  = 0;

    uut.DataMemory.data\_mem\_ff[12]  = 0;

  // REGISTER PRELOAD

  uut.RegBank.reg\_file\_ff[1] = 31'h10010000;

  uut.RegBank.reg\_file\_ff[28] = 31'h10010000;

nop

lw $0, 0($1) 1) First load is not stored, as it points to $zero

lw $2, 32($1) 2) Value from mem[8] 🡪 stored to $2 == 2

lw $4, 36($1) 3) Value from mem[9] 🡪 stored to $4 == 1

add $2, $2, $4 4) $2 = 2+1 = 3 DEP.

and $3, $2, $4 5) $3 = ‘d2 & ‘d1 = ‘d1 DEP.

lw $5, 40($1) 6) Value from mem[10] 🡪 stored to $5 == 5

or $7, $5, $0 7) $7 = ‘d5 | ‘d0 = ‘d5 DEP.

lw $6, 8($1) 8) Value from mem[2] 🡪 stored to $6 == 8

or $7, $7, $6 9) $7 = ‘d5 | ‘d8 = ‘dD DEP.

slt $10, $0, $7 10) $10 = (0 < 7) == 1 DEP.

slt $11, $7, $0 11) $11 = (7 < 0) == 0

addi $31, $0, -1 12) $31 = 0 – 1 == -1

beq $7, $0, FAR 13) BEQ That does not jump 7 != 0

j LABEL1 14) Jump to label 1 BEQ TAKEN HAZARD

LABEL2:

beq $0, $0, LABEL2 15 16) Perpetual jump to itself BEQ TAKEN HAZARD

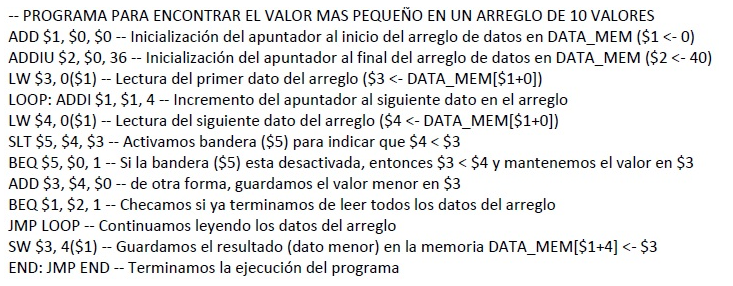
LABEL1:

j LABEL2 15) Jump to label 2 BEQ TAKEN HAZARD

FAR:

addi $zero $zero, 1 // NEVER JUMPS HERE

**Algorithm for finding the smallest value on a 10-value array:**



Actual ASM code:

## FIND LOWER VALUE

## GETTING THE ADDRESS OF THE DATA MEMORY (0x10010000)

## IS PRELOADED

addi $2, $1, 36

lw $3, 0($1)

LOOP:

addi $1, $1, 4

lw $4, 0($1)

slt $5, $4, $3

beq $5, $0, SLT

add $3, $4, $0

SLT:

beq $1, $2, EXIT

j LOOP

EXIT:

sw $3, 4($1)

END:

addi $0, $0, 0

j END

  // -------- LOWEST VALUE ---------------------------

    // STORED PROGRAM

  uut.InstructionMemory.regData[0]  = 0;

  uut.InstructionMemory.regData[1]  = 32'h20220024;

  uut.InstructionMemory.regData[2]  = 32'h8c230000;

  uut.InstructionMemory.regData[3]  = 32'h20210004;

  uut.InstructionMemory.regData[4]  = 32'h8c240000;

  uut.InstructionMemory.regData[5]  = 32'h0083282a;

  uut.InstructionMemory.regData[6]  = 32'h10a00001;

  uut.InstructionMemory.regData[7]  = 32'h00801820;

  uut.InstructionMemory.regData[8]  = 32'h10220001;

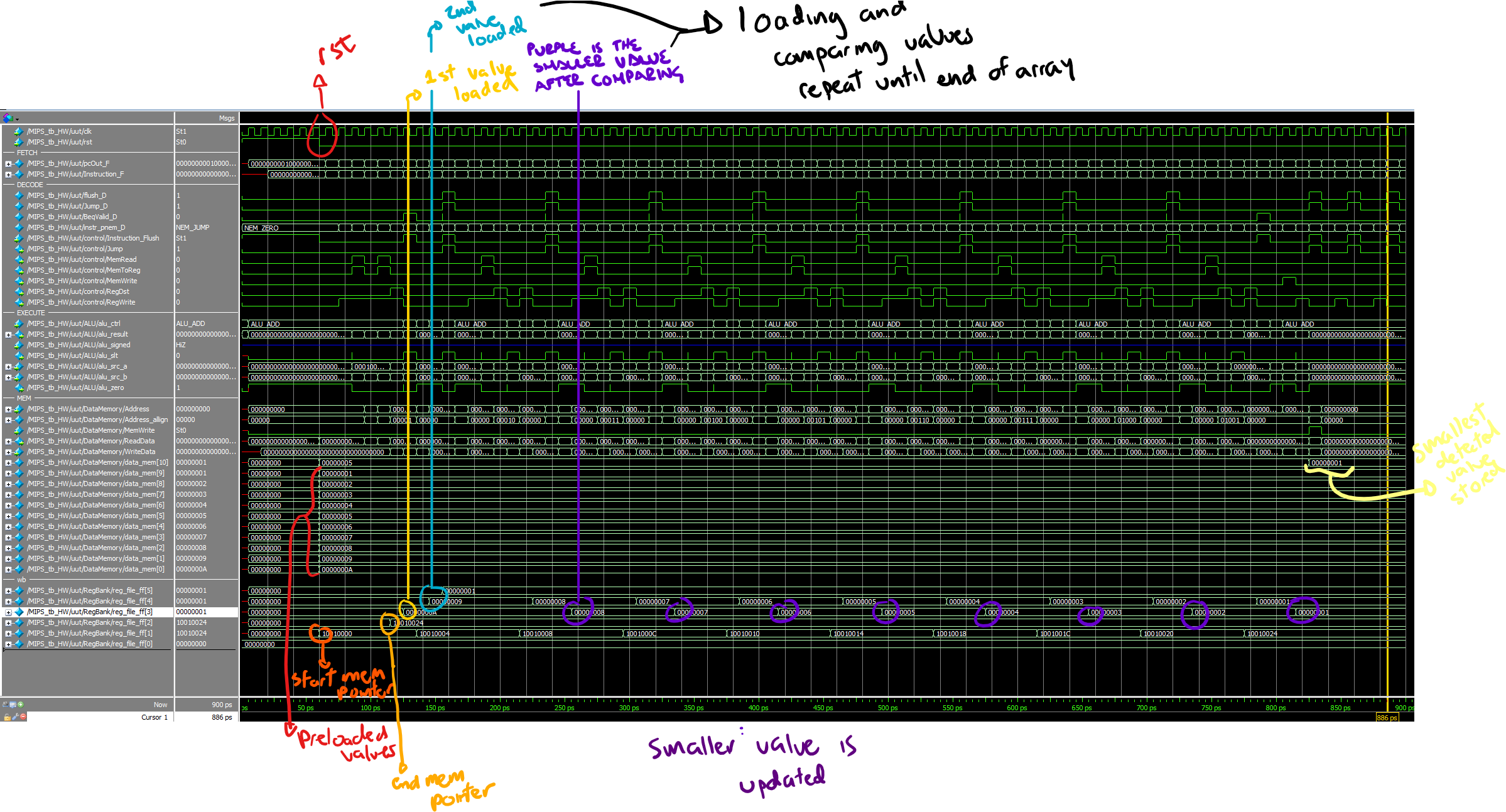
  uut.InstructionMemory.regData[9]  = 32'h08100003;

  uut.InstructionMemory.regData[10] = 32'hac230004;

  uut.InstructionMemory.regData[11] = 32'h20000000;

  uut.InstructionMemory.regData[12] = 32'h0810000b;

  uut.InstructionMemory.regData[13] = 32'h0;



**Multiplier Hazard Validation Program:**

Is also in repo, under the asm directory.

$REPO/asm/smallest\_value.asm

#Basic Init

nop

addi $1, $zero, 0xf

addi $2, $zero, 0x2

addi $3, $zero, 0xff

# Filling pipe with independent multiplications - these are should not be stalled

# Multiplications in this MIPS get stored in register $31

mult $2, $2

mult $1, $2

mult $3, $1

mult $zero, $zero

mult $3, $2

mult $3, $3

# Write port hazard due to B2B

addi $6, $30, 1 # Should be stalled - Write register port is busy

nop

nop

# Stalling instructions with dependency in the destination

# Addi wants to use the destination register of the multiplication

mult $2, $2

addi $6, $31, 1 #Should be stalled -- Result should be diferent than previous addi

add $6, $31, $2

nop

nop

# Write port structural Hazard -- w/ clean pipe

mult $3, $5

nop

addi $5, $3, 1 #This should be stalled - Write register port is busy

add $5, $3, $5

nop

nop

# Hazard in destination port -- The result of the addi should be stored after the multiplication

mult $3, $2

addi $31, $zero, 1 #This needs to be stalled to be correctly written back and stored in registers

add $10, $4, $6

nop

nop

## Independent operation -- this should not be stalled

mult $1, $2

add $4, $5, $6

nop

nop

nop

nop

## Load2use multiply, then store multiply result (double hazard)

lw $1, 4($28)

mult $1, $4

sw $31, 4($28)

# Branch Interruption -- this will be taken, mult should be cancelled/not started

beq $zero, $zero, HERE

mult $2, $31 # This should not finish executing

add $1, $1, $1 #This should not execute

add $1, $1, $1 #This should not execute

add $1, $1, $1 #This should not execute

add $1, $1, $1 #This should not execute

HERE:

addi $2, $2, 1 # Should land here infinitely

nop

j HERE

**Binary:**

  uut.InstructionMemory.regData[0]  = 0;

  uut.InstructionMemory.regData[1]  = 32'h2001000f;

  uut.InstructionMemory.regData[2]  = 32'h20020002;

  uut.InstructionMemory.regData[3]  = 32'h200300ff;

  uut.InstructionMemory.regData[4]  = 32'h00420018;

  uut.InstructionMemory.regData[5]  = 32'h00220018;

  uut.InstructionMemory.regData[6]  = 32'h00610018;

  uut.InstructionMemory.regData[7]  = 32'h00000018;

  uut.InstructionMemory.regData[8]  = 32'h00620018;

  uut.InstructionMemory.regData[9]  = 32'h00630018;

  uut.InstructionMemory.regData[10] = 32'h23c60001;

  uut.InstructionMemory.regData[11] = 32'h00000000;

  uut.InstructionMemory.regData[12] = 32'h00000000;

  uut.InstructionMemory.regData[13] = 32'h00420018;

  uut.InstructionMemory.regData[14] = 32'h23e60001;

  uut.InstructionMemory.regData[15] = 32'h03e23020;

  uut.InstructionMemory.regData[16] = 32'h00000000;

  uut.InstructionMemory.regData[17] = 32'h00000000;

  uut.InstructionMemory.regData[18] = 32'h00650018;

  uut.InstructionMemory.regData[19] = 32'h00000000;

  uut.InstructionMemory.regData[20] = 32'h20650001;

  uut.InstructionMemory.regData[21] = 32'h00652820;

  uut.InstructionMemory.regData[22] = 32'h00000000;

  uut.InstructionMemory.regData[23] = 32'h00000000;

  uut.InstructionMemory.regData[24] = 32'h00620018;

  uut.InstructionMemory.regData[25] = 32'h201f0001;

  uut.InstructionMemory.regData[26] = 32'h00865020;

  uut.InstructionMemory.regData[27] = 32'h00000000;

  uut.InstructionMemory.regData[28] = 32'h00000000;

  uut.InstructionMemory.regData[29] = 32'h00220018;

  uut.InstructionMemory.regData[30] = 32'h00a62020;

  uut.InstructionMemory.regData[31] = 32'h00000000;

  uut.InstructionMemory.regData[32] = 32'h00000000;

  uut.InstructionMemory.regData[33] = 32'h00000000;

  uut.InstructionMemory.regData[34] = 32'h00000000;

  uut.InstructionMemory.regData[35] = 32'h8f810004;

  uut.InstructionMemory.regData[36] = 32'h00240018;

  uut.InstructionMemory.regData[37] = 32'haf9f0004;

  uut.InstructionMemory.regData[38] = 32'h10000005;

  uut.InstructionMemory.regData[39] = 32'h005f0018;

  uut.InstructionMemory.regData[40] = 32'h00210820;

  uut.InstructionMemory.regData[41] = 32'h00210820;

  uut.InstructionMemory.regData[42] = 32'h00210820;

  uut.InstructionMemory.regData[43] = 32'h00210820;

  uut.InstructionMemory.regData[44] = 32'h20420001;

  uut.InstructionMemory.regData[45] = 32'h00000000;

  uut.InstructionMemory.regData[46] = 32'h0810002c;

  uut.InstructionMemory.regData[47] = 32'h0;

Screenshots:

1ST Case:

* 6 mult b2b
* Addi with no dependencies
* This causes a write port dependency.
  + Needs to be stalled.

# Filling pipe with independent multiplications - these are should not be stalled

# Multiplications in this MIPS get stored in register $31

mult $2, $2

mult $1, $2

mult $3, $1

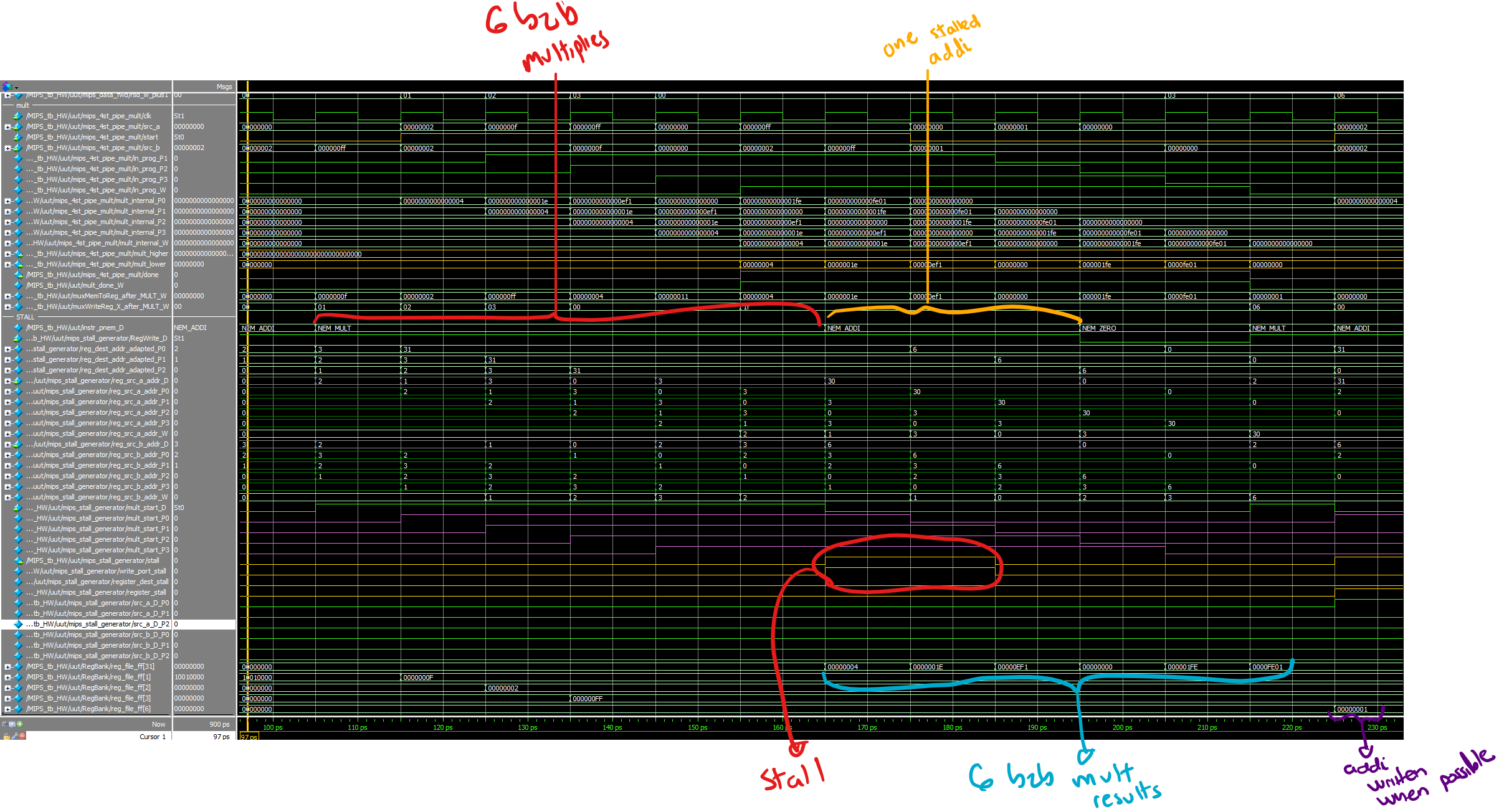
mult $zero, $zero

mult $3, $2

mult $3, $3

# Write port hazard due to B2B

addi $6, $30, 1 # Should be stalled - Write register port is busy



2nd case:

* Mult op
* Followed by addi with a dependency.
  + SRC register is the multiplication’s DEST register.
  + Should be stalled

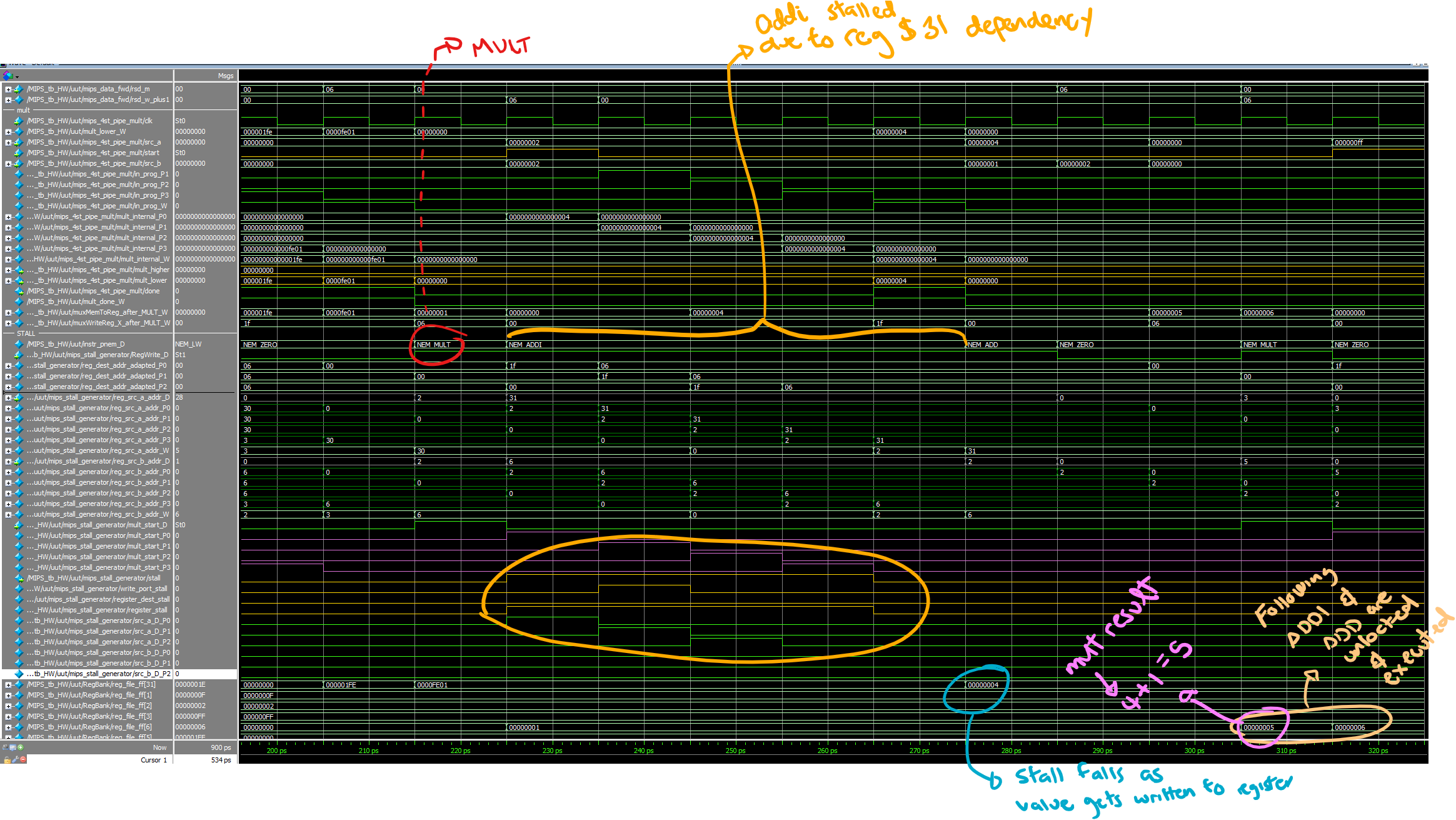
# Stalling instructions with dependency in the destination

# Addi wants to use the destination register of the multiplication

mult $2, $2

addi $6, $31, 1 #Should be stalled -- Result should be diferent than previous addi

add $6, $31, $2



3rd case:

* Isolated write port structural hazard
  + Multiply
  + Nop
  + Addi
    - This add will crash when trying to use write port -- structural hazard.

# Write port structural Hazard -- w/ clean pipe

mult $3, $5

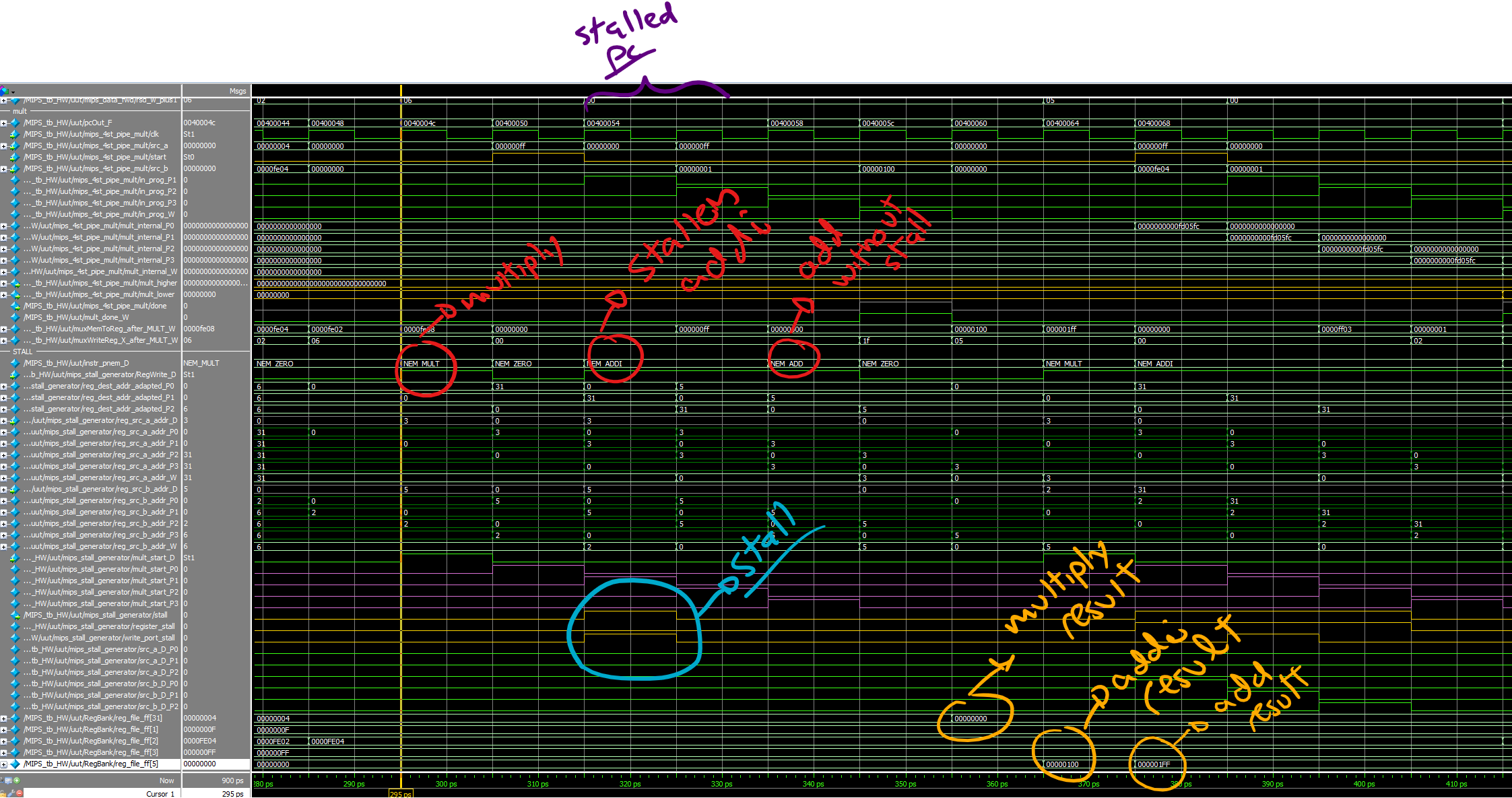
nop

addi $5, $3, 1 #This should be stalled - Write register port is busy

add $5, $3, $5

nop

nop



4th case:

* Hazard in destination port
  + Multiply
  + Addi
    - Writes to same destination register as multiply
    - Need to stall it, because this is the result that needs to stay in the register.

# Hazard in destination port -- The result of the addi should be stored after the multiplication

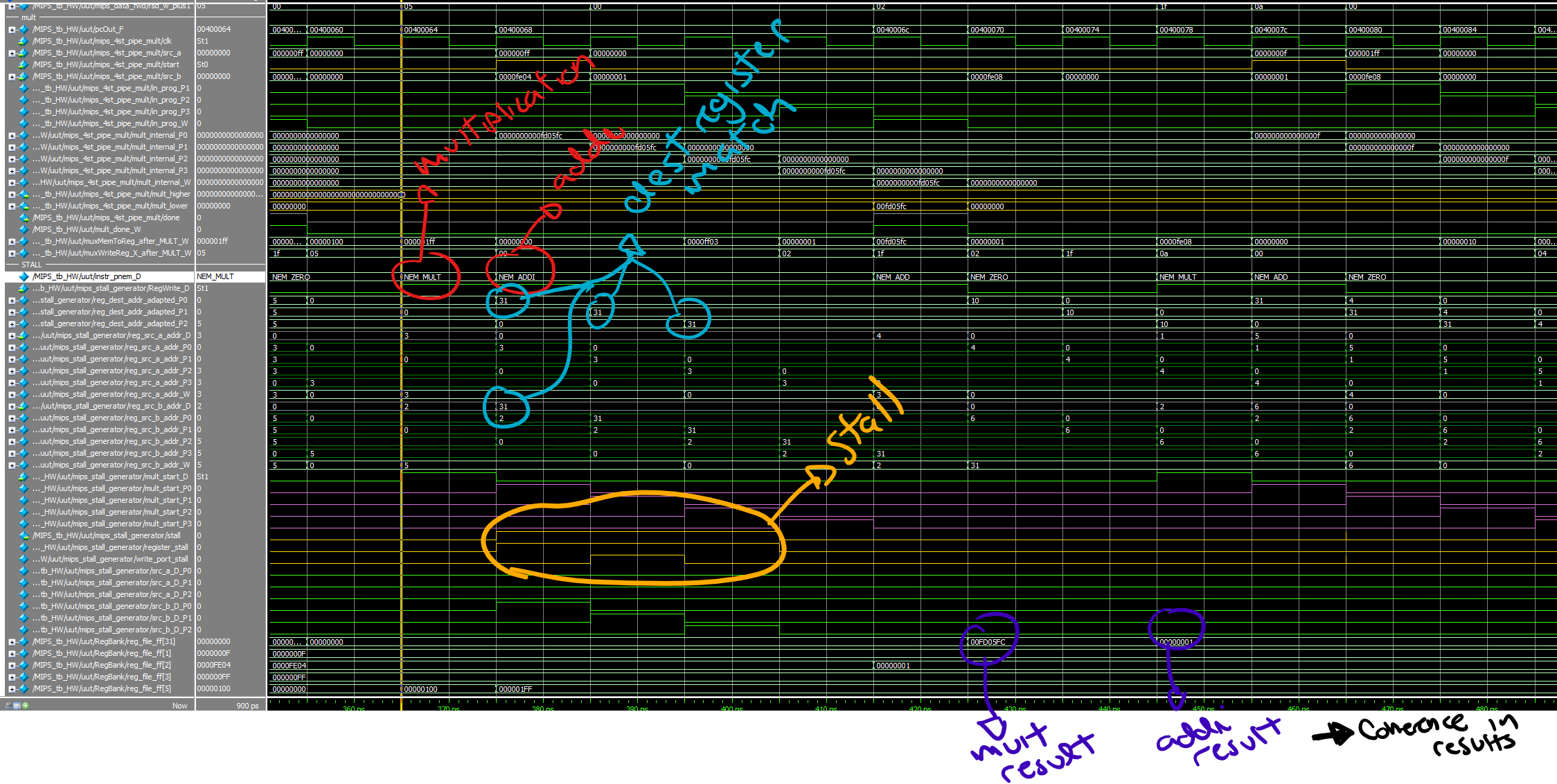
mult $3, $2

addi $31, $zero, 1 #This needs to be stalled to be correctly written back and stored in registers

add $10, $4, $6

nop

nop



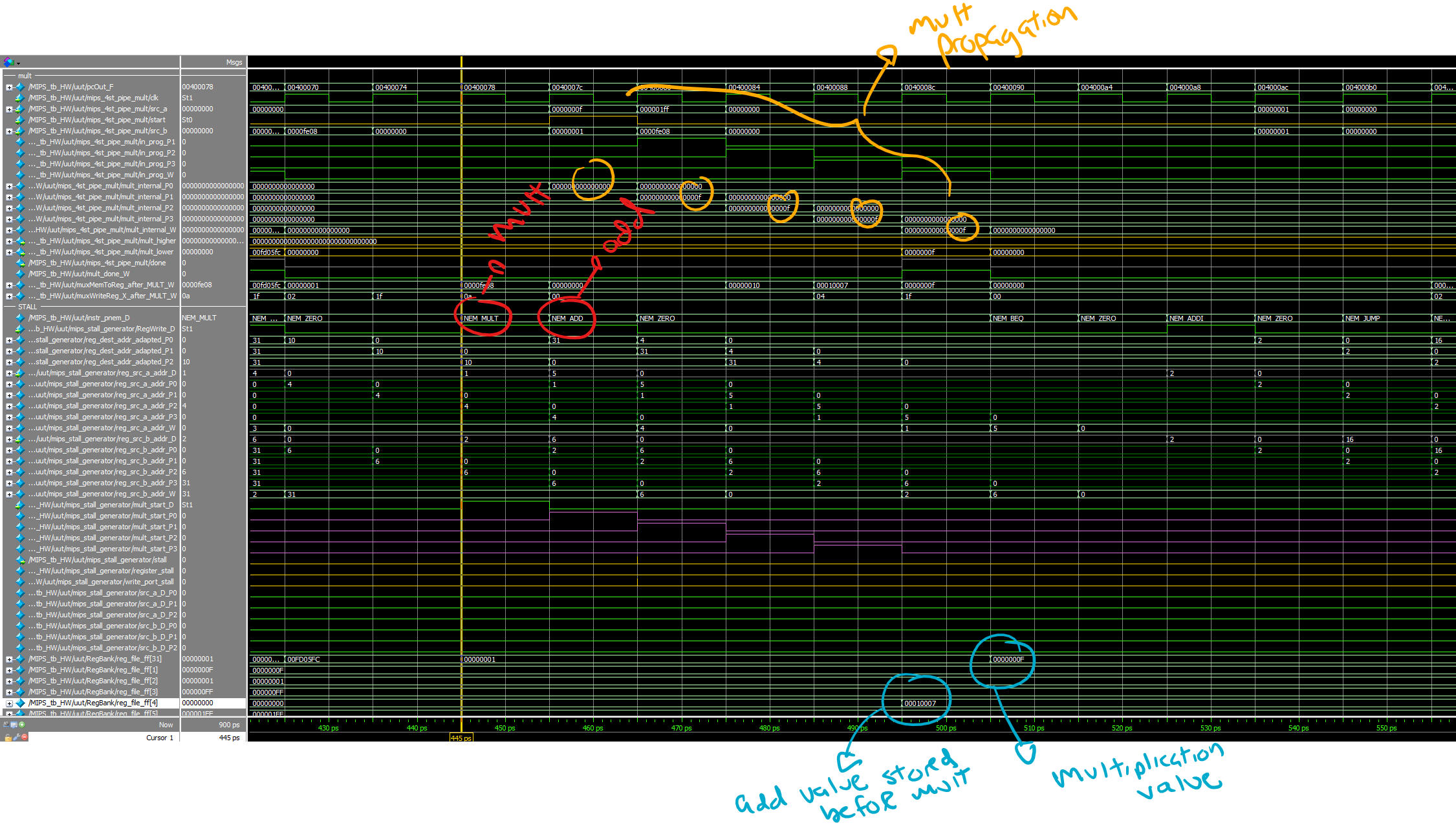
5th case:

* Independent operations
  + Mult
  + Add
  + Not stall needed
  + Add will be stored in register before the multiplication

## Independent operation -- this should not be stalled

mult $1, $2

add $4, $5, $6

nop

6th case:

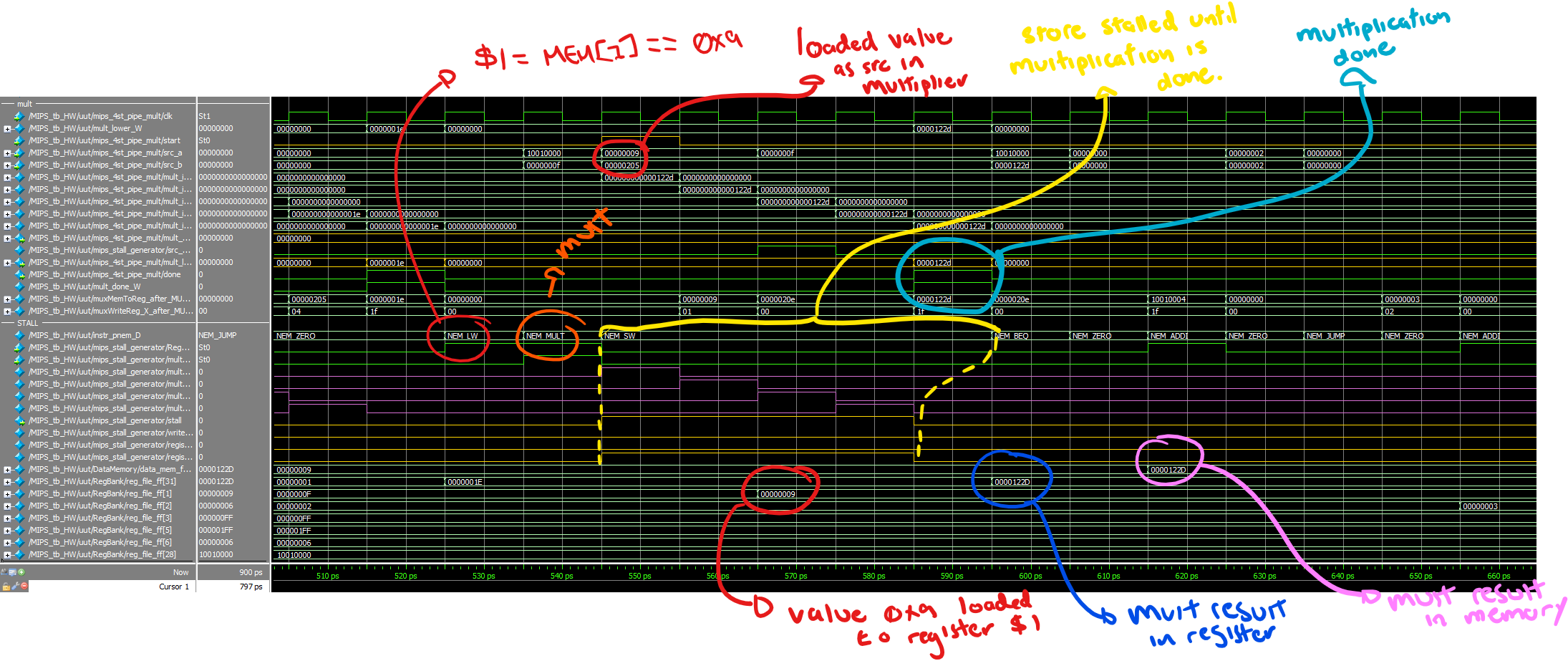
* Double case
  + Load from memory containing value to multiply next
  + Storing the value just multiplied

## Load2use multiply, then store mult value

lw $1, 4($28)

mult $1, $4

sw $31, 4($28)



7th case:

* Branch taken interrupting not taken mult
  + Branch
    - This is taken
    - Predict not taken
  + Multiply
    - Not taken
    - Needs to be cancelled

# Branch Interruption -- this will be taken, mult should be cancelled/not started

beq $zero, $zero, HERE

mult $2, $31 # This should not finish executing

add $1, $1, $1 #This should not execute

add $1, $1, $1 #This should not execute

add $1, $1, $1 #This should not execute

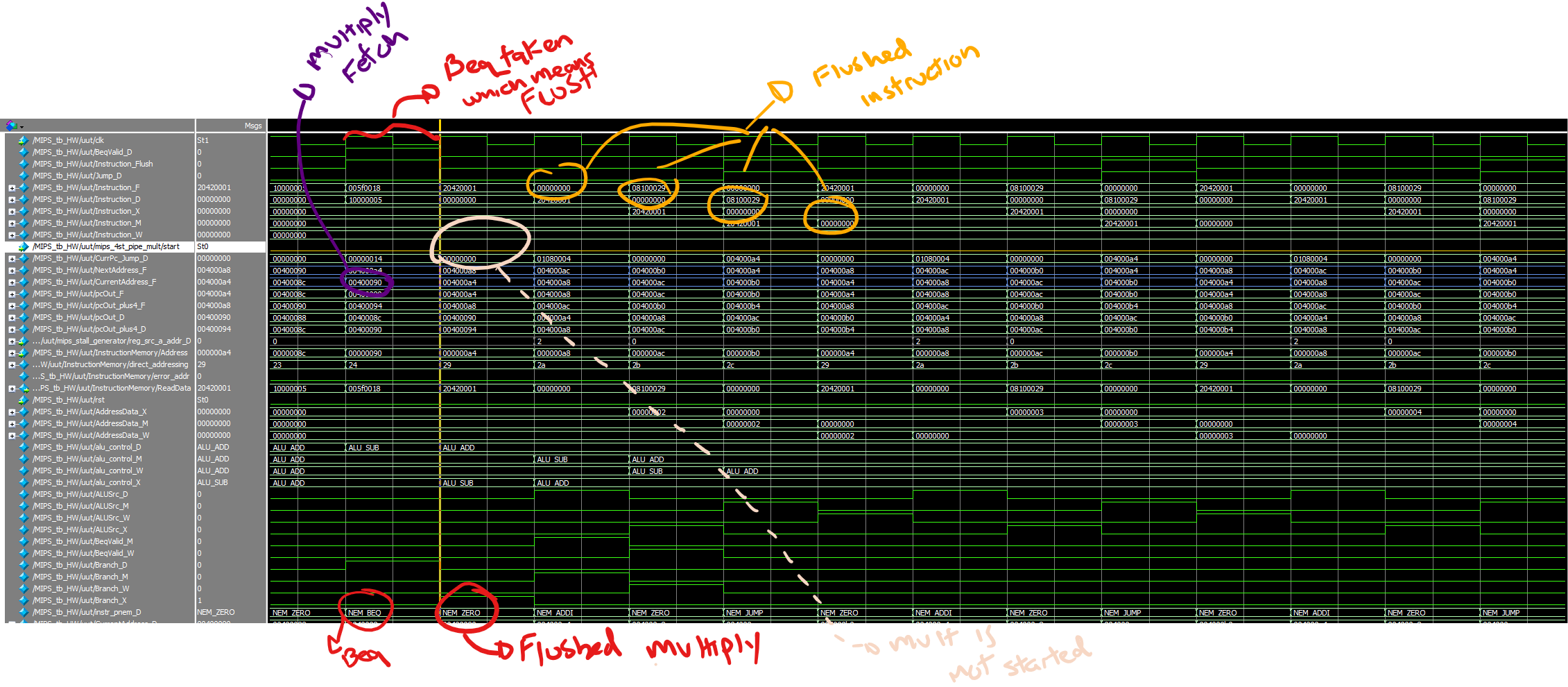
add $1, $1, $1 #This should not execute

HERE:

addi $2, $2, 1 # Should land here infinitely

nop

j HERE



To run

Running the testbench:

* **COMMENT AND UNCOMMENT THE DESIRED PROGRAM TO EXECUTE**

Create a ModelSim project with all the files in the $REPO/src

Compile all the files

Open the waveform of the file MIPS\_tb2.sv

$REPO/src/MIPS\_tb2.sv

Note: The module name is called MIPS\_tb\_HW



Open the do file:

$REPO/ModelSim/wave2.do

Modifying the values of the test:

Open and edit the file

$REPO/MIPS\_GIT/src/MIPS\_tb2.sv